COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

(18) AFOSR

(19) TR-81-0661

Technical Report TR-1065 (11) ~~July 1981~~
[redacted]R-F49620-80-C-0001
(15)

(6) Error Propagation and Elimination in Computer Programs*

(12) 14

(16) 2304

(17) A2

(10) Larry J. Morell ~~and~~ Richard G. Hamlet

(9) Technical rept.

(14) TR-1065

Abstract:

The current literature in program testing is surveyed. A strategy is proposed for eliminating categories of errors from programs. Errors may be classified as functional (an incorrect input-output pair) or structural (an incorrect statement). An error is eliminated if a successful program execution for a given input implies the program could not contain the error. A "creation condition" guarantees that a structural error affects the program's computation. A "propagation condition" guarantees that the effect produces a functional error. An error is eliminated whenever a computation satisfies both the creation and the propagation condition and produces correct output.

---

*Research supported in part by the Air Force Office of Scientific Research, grant number [redacted]-F49620-80-C-0001.

411 22

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 81 -0661 / 2. GOVT ACCESSION NO. AD A105437	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ERROR PROPAGATION AND ELIMINATION IN COMPUTER PROGRAMS	5. TYPE OF REPORT & PERIOD COVERED Technical Report
	6. PERFORMING ORG. REPORT NUMBER TR-1065
7. AUTHOR(s) Larry J. Morell and Richard G. Hamlet	8. CONTRACT OR GRANT NUMBER(s) -F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park, Maryland 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB DC 20332	12. REPORT DATE July 1981
	13. NUMBER OF PAGES 46
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)	
18. SUPPLEMENTARY NOTES	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) mutation testing, function testing, reliability, error propagation, error elimination	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The current literature in program testing is surveyed. A strategy is proposed for eliminating categories of errors from programs. Errors may be classified as functional (an incorrect input-output pair) or structural (an incorrect statement). An error is eliminated if a successful program execution for a given input implies the program could not contain the error. A "creation condition" guarantees that a structural error affects the program's computation. A "propagation condition" guarantees that the effect produces a (over)

DD FORM 1 JAN 73 1473

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (cont.) functional error. An error is eliminated whenever a computation satisfies both the creation and the propagation condition and produces correct output.

1. Introduction

Rarely has a developing field rapidly attained a unified understanding of itself; program testing is no exception. Three problems must be addressed if progress is to be made.

(1) How can the quality of test data be measured?

(2) How can the quality of a testing strategy be measured?

(3) What is an appropriate paradigm for program testing?

A positive answer to the first question would provide confidence in the results of testing a single program. For now, the tester can merely cite a few statistics (percentage of paths executed, percentage of branches executed, etc.). But what is the value of executing 80% of the paths? In what sense, if any, is it better to execute 1000 test cases rather than 100? Without an underlying theory statistical claims are dangerous, because they can lull the tester into a false sense of security.

Answering the second question does not automatically answer the first; a good strategy may sometimes produce a bad test set. The characteristics of a good strategy could guide researchers into more profitable areas. It is entirely possible that strategies must be specialized for different program classes. How then can the various strategies be compared? Does it even make sense to compare strategies that cannot be used for the same program? The ability to determine a test's quality does not necessarily imply the ability to determine a testing strategy's quality; this would require inferring the quality of a testing strategy from its results on a finite number of applications. This, in essence, is using testing to measure the quality of testing, a rather dubious approach at best.

The third question suggests that testing is more than searching for hidden program errors. Most strategies use what may be called an error discovery paradigm; i.e., the ultimate goal of a testing strategy is to generate inputs that show a program is incorrect. When a program executes successfully, the fact is recorded and the search continues for an input that will reveal an error. Thus, the error discovery paradigm only allows the conclusion that a program is correct on its tested domain. This paper suggests that an error elimination paradigm is more appropriate. An error is eliminated if a successful program execution for a given input implies the program could not contain the error. Such an approach allows the conclusion that specific errors are not contained in a program.

This paper discusses one way of eliminating errors from programs through the use of creation and propagation conditions. A creation condition guarantees that a potential error in the code affects the program's computation. A propagation condition guarantees that the effect produces an output error. If the output is correct, the potential error did not occur, and thus can be eliminated from the program. Section 2 of this paper surveys the best known functional and structural testing methodologies. Section 3 discusses the results currently known from testing theory. Section 4 presents reliability theory and develops it in the context of error elimination as the goal for testing. Section 5 introduces an error elimination strategy for testing programs; it is based upon the concept of "error propagation." The final section proposes areas in which further research seems promising.

2. Survey of Dynamic Validation

2.1. Dynamic analysis

Dynamic analysis [How78] involves the execution of a given program with specific test data. The output is compared with the specification to decide correctness. Test data selection may be based upon the actual code or upon the specifications. The former case is termed "structural testing" since the structure of the program is considered in the test data selection. The latter case is termed "functional testing" because only the input-output behavior is considered in choosing test data.

2.1.1. Structural Testing

The goal in structural testing is program coverage. If the code of a program can be sufficiently "exercised" (or covered) it seems reasonable to conclude that any incorrect code will manifest itself, thus revealing the presence of an error. Miller [Mil74] and Howden [How78] suggest the following two structural coverage criteria:

(1) Statement coverage - Every statement should be executed. It is unreasonable to expect that unexecuted code will perform correctly when executed.

(2) Path coverage - Every path in the program should be executed. "Path" is defined to be any possible flow of control through an uninterpreted flowchart. Thus a path from a given flowchart may not in fact be executable due to the particular conjunction of conditions "guarding" the path. Howden calls such a path _infeasible_ [How76].

Clearly path coverage is impossible for any program containing a loop with a run time determined exit condition, since each repetition of the loop determines a new path. Various approximations to path coverage are suggested to reduce the problem to manageable size. Among these are branch testing and path equivalence classes.

(1) _Branch testing_. One approximation to path coverage is to ensure that all potential branches are executed. "Potential branches" has been alternately defined to mean "the potential outcomes of a given conditional" or "the means by which those outcomes can be

obtained." The difference arises in compound conditional such as A v B where the potential outcomes would merely require A v B to evaluate in one case to T and in another case to F. The true outcomes may be obtained from several different cases such as A = T, B = F and A = F , B = T as well as A = T, B = T. A simple resolution of the difference is to require branch testing to be performed on modified programs in which all compound conditional are expanded into simple conditionals.

(2) <u>Path equivalence classes</u>. For the infinite set of paths in a given program, paths may be equated which share various structural criteria. For example, level testing equates paths that have the same depth of nesting within a program as determined from the static code [Mil74]. This technique aims at testing nested paths, thereby guaranteeing coverage of all decision-to-decision paths in the program. A corresponding dynamic path equivalence relation equates paths containing at most n iterations of all loops.

The inadequacy of structural testing is shown by the following incorrect solution for computing the maximum of a and b.

```
if a > b
  then   max := a
  else   max := -a
```

The test {(a=1,b=-1), (a=-1,b=1)} satisfies all the structural criteria given, (all statements are executed, all branches are taken, all paths are executed), yet, the error is not evidenced for this particular test.

As a result of pernicious examples like this, more refined structural criteria have been proposed which require more detailed differentiation by the test. These areas may be broadly defined as mutation testing [DeM78] [Bud80], function testing [Fos78] [How80], and domain testing [Zei80]. Each of these makes additional assumptions about the nature of the program design, structure, or execution behavior. With these assumptions a greater refinement of test cases is possible, resulting in a greater "exercise" of the program.

Mutation testing [DeM78] assumes the "competent programmer hypothesis," namely that a competent programmer under normal conditions will produce code that is close to being the correct code. Labeling the programmer's code as P and the correct code as P*, it is reasonable to assume that relatively few syntactic changes in P will result in P*. Alternately, P* has many "mutants" that are quite close syntactically. A test set is considered reliable if it differentiates P* from all of its mutants. A mutant is differentiated when it executes incorrectly on a given test set, in which case the mutant is said to be "killed." If all reasonable mutants are killed by a given test set, correct operation on that test set implies the program contains no "unreasonable" errors. If the competent programmer hypothesis holds, the test set is reliable since competent programmers produce only reasonable mutants.

To limit the number of mutants, it is necessary to restrict the types and combinations of changes allowed. Common restrictions are to allow replacing expressions with limited size expressions, to disallow inserting of arbitrary statements, and to disallow making arbitrary changes in the flowgraph. It is cogently argued [DeM78] that a test set which kills single mutants will also kill double mutants. Empirical studies [Bud80] involving mutation testing have shown it to be quite effective as well as quite expensive, since a large number of mutants must be generated and executed. Hamlet's system [Ham77] reduces this time by executing compiled code up to the chosen point of mutation, and then successively trying each mutant. Each system faces two theoretical problems, namely, what happens when the mutant does not halt within a specified period of time, and what happens when the program does halt with correct output. In the first case, an arbitrary time limit must be invoked, usually a function of the running time of the original. In the second case, a human must ultimately intervene. If the mutant is not the same as the original (as in the case of an algebraic simplification), then the test set must be augmented. The process begins again until all mutants are killed (or shown equivalent to the correct program).

Foster has proposed a method that may be called function testing in which he gives criteria for testing specific program constructs for typical errors [Fos78]. Howden has generalized this to make test cases sensitive to potential errors in any of the primitive semantic functions supported by a programming language. For instance, consider a language in which each variable has two associated functions, STORE (var, value) and RETRIEVE (var, value). The variable's RETRIEVE function is invoked whenever the variable must be evaluated; the variable's STORE function is invoked whenever the variable is assigned a value. If a mutation occurs that substitutes one variable for another variable in an expression, then the wrong RETRIEVE function would be invoked when that expression is evaluated. If the test set establishes at the mutation point different values for all variables, then the wrong RETRIEVE function would introduce an incorrect value into the evaluation of the expression. If the effect of this incorrect value propagates to the output then the error will be <u>manifested</u>. This test set is in some sense reliable for discovering errors involving the use of a wrong variable in an expression. Howden extends this to considerably more complex functions commonly occurring in a programming language. The method can potentially eliminate an entire category of mutants on a single execution. Its weakness lies in not guaranteeing that potential errors manifest themselves.

<u>Linear domain testing</u> [Zei80] is an application of theoretical ideas on path testing given by Howden [How76]. Each path can be uniquely characterized by a subset of the input space called the <u>path domain</u>. A program contains a <u>domain error</u> if an incorrect path is followed for an input and produces incorrect output. An incorrect computation along a path is called a <u>computation error</u>. Domain testing therefore is a version of path testing. A <u>linearly domained program</u> P satisfies the following:

(1) An input cannot follow an incorrect path and produce correct output.

(2) No paths are missing from P.

(3) The input space for P is continuous.

(4) P contains no compound predicates.

(5) Adjacent domains compute different functions.

(6) Each predicate in the program is a linear transformation of the program inputs.

Certain predicate errors may not be detectable by testing a particular path. For example, it is impossible to determine if "3" is the correct constant in the predicate "x + 3*y < 0", for a path in which y has a constant value of 0. Such situations may arise when a path assigns 0 to y ("assignment blindness") or selects only 0-valued y's ("equality blindness"). Since assignment and equality blindness are characteristics of the path up to a predicate, no amount of testing of the path can eliminate the possibility of certain errors in the predicate. Thus, every path containing a predicate implicitly defines a set of errors that cannot be eliminated from the predicate by testing that path. The errors in a given predicate that cannot be eliminated by testing a collections of paths is the intersection of all the non-detectable errors determined by each path in the collection. Thus, testing an additional path is useful only if the non-detectable errors for the new path does not contain the intersection of all the non-detectable errors for the paths already tested.

In thr case of a linearly domained program, both paths and predicates can be modeled as linear transformations. Let C be the transformation for a path up to a predicate, let T be the transformation for the predicate, and let T' = T + E be an erroneous version of T. The transformation T' may not detectably different from T' because TC = T'C, or equivalently, EC = Z (Z is the zero vector). Solving EC = Z for values of E yields those predicate errors which are not detectable due to assignment blindness. Predicate errors may also remain undetected whenever EC <> Z but ECv = 0 for all v in the path domain. Solving this equation for E yields those predicate errors which are not dectable due to equality blindness.

2.1.2 Functional Testing

In functional testing test cases are selected to exercise the specifications rather than the code itself. This is sometimes termed a "black box" approach to testing since the code is ignored as a source of information for selecting test data. The program's function is the only concern; if the program satisfies the specification it is correct and coverage criteria are unnecessary. Of primary concern are the special values for each input variable given by the specification. Test points are selected to ensure that values are input for both extremal and non-extremal points as well as special values of every variable. This quickly results in a combinatorial explosion, controlled by partitioning and refining the overall requirements for the code. Partitioning associates inputs that are closely relationed to one another; refining associates particular functions and the code that implements them. Howden [How80b] provides an excellent overview of functional testing.

The most general specification available for functional testing is the requirements document that specifies overall system operation. Testing requirements involves selecting test points that aim at determining overall satisfaction of the system goals. Details of how the function is computed are ignored; an attempt is made to handle the different combinations of possible input categories. Consider, for example, a file system. There may be requirement that a COPY does not destroy the original file. Such a requirement may be tested without regard for where or how files are stored. In designing the system, decisions are made on how to represent a particular file type. These decisions imply that certain functions may be necessary to implement the COPY operation; these are termed "design functions." Testing of these individual functions may be done in the same manner as the testing of the COPY requirement, but on a smaller scale. Even more detailed design functions may be specified at a lower level. In this manner, the combinatorial problems are somewhat decreased. By identifying various abstractions that are present in the input data it is possible to further reduce the number of combinations.

2.2. Symbolic Execution

Before leaving this survey of testing methodologies, it is appropriate to comment upon a hybrid between testing and formal verification called symbolic execution [How77] [Cla77] and [Han76]. Formal verification requires a proof of various mathematical properties to demonstrate correctness; symbolic execution aids in the proofs of these properties by allowing execution of the program with symbolic data. This is one step beyond data flow systems such as DAVE [Ost76] in which the program is abstracted into a flowgraph with movement of data along the paths. The entire semantics of the programming language must be at hand to enable complete interpretation of the program during symbolic execution. This enables the construction of path conditions (the sequence of decisions made along a path) which can be an aid in documenting the program and in determining actual test data that can satisfies the path condition. Formal verification is aided in providing a description of the output in terms of the input (and possibly constants) which then need to be shown to satisfy the output condition. The input condition aids in determining what branches may be chosen, either beforehand as in DISSECT [How77] or interactively as in EFFIGY [Han76].

Symbolic execution systems which attempt to deduce the value of conditionals are only as strong as the theorem provers upon which they rely. The inability of a theorem prover to decide the value of a conditional does not guarantee that the value cannot be decided. Thus, human input may be required more frequently than necessary. Similarly, having the path condition determined is of little value if test cases cannot be automatically generated to satisfy the condition. Since such generation is impossible (as discussed in the next section of this paper), the system must again rely upon human input. The path condition is frequently so complex, that it is often easier for a person to generate the test data from the code rather than the condition.

3. Development of Testing Theory

The development of testing theory has followed mostly two directions, one of general unsolvability and one of solvability over particular classes of programs. General unsolvability results [Hen77] rely heavily on recursive function theory and deal with automatic generation of test sets. With the general results rather dismal, specific exceptions have been investigated. The search for classes of programs in which testing is tantamount to formal verification is an open area of research.

General unsolvability results in testing theory ultimately lie close to the heart of recursive unsolvability, the halting problem. Formal proofs of the results in this section may be found in many excellent sources [Ham74] [Hen77]; the presentation here will be from a testing viewpoint. First, we need some notation and a few simple definitions, as taken from [Lin79].

Notation: If P is a Program then [P] denotes the function that P computes. The output of P on input x may be written as [P](x), if [P] is defined for input x. Dom([P]) denotes the domain of [P]. "In" designates set membership and "*" designates set intersection.

Definition A specification S is a set of ordered pairs satisfying the following:

a. S is recursive.

b. dom(S) is recursive.

Definition A program P is said to be correct with respect to a specification S iff

dom ([P] * S) = dom (S)

Definition A test set is a subset of the domain of a specification.

Definition A program P satisfies a specification S on a test set T iff

For All x In T, [P](x)is defined and (x, [P](x)) In S.

The following classic theorems from recursive function theory are included for completeness sake. For proofs see [Hen77] or [Ham74].

<u>Theorem</u> (Halting Problem) Let P_1, P_2, ... be an effective enumeration of all programs (say by their lexical order). There does not exist a program P satisfying the following:

$$[P](x) = \begin{cases} 1 \text{ if } [P_x](x) \text{ is defined} \\ 0 \text{ if } [P_x](x) \text{ is not defined} \end{cases}$$

<u>Theorem</u> (Program Equivalence Problem) There does not exist a program P satisfying the following:

$$[P](x,y) = \begin{cases} 1 \text{ if } [P_x] = [P_y] \\ 0 \text{ otherwise} \end{cases}$$

We obtain almost immediately from the above theorems the following result:

<u>Corollary</u>

There does not exist a program that generates or recognizes a test set T that satisfies any of the following properties (for all programs P, specifications S, paths p, statements s, expressions e, and values v):

a. P satisfies S on nonempty T
b. Path p of P is executed by T
c. Statement i of P is executed by T
d. Expression e in P evaluates to value v
e. P satisfies S on a nonempty subset of T

The results from above lead to a Murphy-like rule for the results of testing theory, namely, if a desired result is powerful and generally applicable then it cannot be obtained. Since weaker results are not usually desired, to maintain strength it is necessary to reduce applicability. Hence, whereas the corollary gives a gloomy general forecast, for specific classes of programs and specifications all the results are obtainable. Three prominent examples, [Bud80], [Tsi70], and [How78b], completely characterize the program function by a finite set of tests and a few restrictions about the program structure.

Early work that has bearing upon testing programs from a particular class comes from complexity theory based on the LOOP hierarchy [Mey67] and further analyzed by Tsichritzis [Tsi70]. Briefly, a loop program consists of assignment statements

```
<assign> ::= <var> := <exp>
<exp>    ::= <var> | <var> + 1 | 0
```

and loop statements

```
<loop>   ::= LOOP <var> <assign>+ END
```

When control reaches a loop statement the <var> is evaluated to a non-negative value and the list of assignment statements is then executed that number of times. Arbitrary nesting of loop statements is allowed. Loop_0 is exactly that class of LOOP programs with only assignment statements. LOOP_i programs $(i>0)$ are LOOP programs in which the maximum nesting level is i. Thus, LOOP_{i+1} syntactically contains all LOOP_i programs. Meyer and Ritchie [Mey67] have shown that LOOP_{i+1} properly contains LOOP_i programs semantically as well. Thus there are some LOOP_{i+1} program functions that are not computable by LOOP_i programs. Furthermore, the infinite union of functions computable by the LOOP programs is exactly the class of primitive recursive functions and thus the hierarchy of functions computed by LOOP programs forms a hierarchy of primitive recursive functions.

Tsichritzis [Tsi70] investigated the first two levels of the LOOP hierarchy to show that LOOP_1 programs correspond to a subclass of primitive recursive functions called simple functions. His result for testing theory is that a finite set of input-output pairs uniquely determines a simple function. He provides an upper bound on the size of the test set which is computable from the simple function. Hence the size can be functionally related to the structure of the LOOP_1 program since the determination of the simple function computed is mechanical. The significance is that LOOP_1 forms a class of programs which has an algorithm for generating a test set that proves the program correct.

Two instances of program classes for which the above corollary has a solvable counterpart are given in [How78b] and [Bud80]. The first class is the set of programs characterized by the functions they compute, multinomials. The second class is a subset of LISP programs that satisfy a particular recursive schema. The techniques used to generate data for these classes are not readily extendible to other program classes because both rely upon the mathematical properties of the functions being computed.

4. Reliability Theory

In attempting to provide a firmer foundation for testing (and to allow it to be called a "theory") a reliability theory has been developed for testing computer programs. This theory encompasses the results of the previous section, and provides a framework for evaluating the various ad hoc testing strategies mentioned earlier in the paper by relating the notion of correctness to that of thoroughness of a test set. Since test sets are essentially finite (excluding symbolic evaluation), a reliable test set must somehow capture the essence of the program on a finite domain. The results from the previous section certainly imply that such sets cannot be algorithmically constructed or recognized except for certain classes of programs. Reliability theory has therefore concentrated on ways in which test sets can be identified for particular classes of programs.

4.1. Early Attempts

Gerhart and Goodenough first attempted to provide a theoretical basis for testing [Goo75]. A test selection criterion C is said to be reliable if and only if all sets that satisfy the criterion either prove the program incorrect (by failing to meet the specifications) or satisfy the specification. A test selection criterion C is said to be valid if and only if for every error point there is a test set that satisfies the criterion C and proves the program incorrect. From these two definitions, Gerhart and Goodenough prove their fundamental theorem, namely, if a reliable test that satisfies the specifications of a program is also valid, then the program is correct. Thus, the job of the tester is to demonstrate that a given criterion is both reliable and valid. Thereafter, one successful execution on a test set satisfying the criterion proves the program. In some cases it is trivial to prove either reliability or validity, but rarely is it trivial to prove both. In fact, as shown by [Wey80], if a test selection criterion is not valid it must be reliable and if it is not reliable it must be valid. Indeed, if C is an invalid criterion, then there is a point for which the program is wrong and no test set discovers this. Hence, all the test sets imply the program might be correct and therefore the criterion is reliable. If the criterion is not reliable, then some of the test sets satisfying the criterion disprove the program. Thus, for every point there is a test set that proves the program incorrect and the test selection criterion C is therefore valid.

In contrast to this thicket of intertwined definitions, Howden [How76] and others have espoused the following definition of reliability:

Definition A test set is reliable for a program P with respect to a specification S iff

P satisfies S on T ==> P satisfies S on dom(S).

The distinction made between reliable and valid are effectively combined into one notion that still allows correctness to be concluded, but at a rather strong price. The cost is found in having to verify that the correctness of the program does follow from its correctness on a finite domain. Howden analyzed path testing in the light of this definition and showed that rather strong assertions must be proved about the program if path testing is to be reliable. Categorizing errors into computation errors (incorrect computation on a given path), domain errors (incorrect path selection), and case errors (missing paths), he was able to show sufficient conditions under which path testing is reliable for two of these errors, assuming compound errors do not occur. The results are as follows:

(1) Computation errors - All members of the path domain for a path containing the error produce incorrect output.

(2) Domain errors - The path domain for the correct and incorrect program share no points in common. Furthermore, the computed function along each path is assumed to be different. This prevents an input from following an incorrect path and still producing the correct output.

(3) Case errors - Howden incorrectly identified these with domain errors, resulting in path testing being reliable for case errors iff useless code exists in the program text.

As can be seen from Howden's results, even assuming that all paths can be tested, reliability is simply too strong a requirement to determine by testing.

4.2. Error Reliability

Weakening the notion of reliability either requires narrowing the class of programs that will be considered or reducing the requirements of correctness. Linear domain testing [Zei80] is an example of the former and mutation testing [DeM78] is an example of the latter. Recently Howden [How80], Foster [Fos78], Ostrand [Wey80], and Weyuker [Wey81] have proposed methods which can be labeled error-based testing strategies. The goal is to demonstrate the absence of certain predefined errors rather than (necessarily) the correctness of the program. Test data is selected to enable errors, if present, to be revealed [Wey80], provided the execution of the program does not prevent an error from being manifested. Thus, error-based testing is an example of reducing the requirement of correctness to weaken the notion of reliability. The following is a definition of modified reliability:

Definition 7 A test set T is E-reliable (Error reliable) for a program P and specification S iff

P satisfies S on T ---> P contains no errors of type E.

It should be noted the concept of error type used in this definition is as yet undefined. In the next section "error" is shown to have two distinct usages, namely to reflect the incorrect operation of the program (a functional error) or to pinpoint the location of the error in the code (a structural error).

4.3. Errors

To gain a deeper understanding of various testing methodologies it is necessary to understand each methodology's concept of error. There are two general vantage points from which errors may be approached, one structural and one functional. In the structural approach an error is considered to be associated with the text of the program, for example, an incorrect conditional that causes some inputs to follow an undesired path. In the functional approach an error is a program-computed input-output pair not satisfying the specifications. In this approach no mention is made of how the output is computed. The difference between the two is evidenced when an input follows an undesired path but produces the correct output. In this case the program has a structural error but a functional error has not been manifested. (It must be the case however that a functional error can be manifested on some other input, or the "error" is not one at all.) Both approaches to error have their advantages and disadvantages and a corresponding range of applicability.

Howden [How76] uses a structural concept of error, in that an error is within a particular program and hence can be spoken of as being a particular expression, within a particular statement, on a given path, etc. Such a structural approach is intuitively satisfying since it emphasizes that incorrect operation of a program ultimately lies in some portion of the program text. Correcting the error therefore naturally translates into transforming the program text. Hence to identify that portion of the program as an error seems natural. This approach has two deficiencies, especially when the concept of errors is used to compare various testing methodologies. First, a structural approach to error definition is more applicable to procedural rather than functional languages, since in the former the location of a given error provides considerably more information than the latter (e.g. the type of the expression values, possible paths, etc). In self modifying languages such as LISP and SNOBOL, statements may be executed which do not even exist in the source code. Second, a structural approach makes the correspondence between specification and correctness difficult to state. For indeed, it may be quite clear that a program has failed to meet a specification by, say, terminating with incorrect output for a valid input. Yet, it is inappropriate to speak of "the" program error since such an error may actually involve the compound result of several statements, none of which is wrong in and of itself, but all are wrong as a whole.

A concept of error that avoids the above problems with the structural approach lies in the operation rather than the structure of the program. Such an approach may be termed "functional" because it deals with the meaning of the program as expressed by its input-output behavior. In a functional approach an error is associated with the input-output behavior of the program as determined by the specifications. An error occurs when a given input produces an incorrect output; such an input is labeled as being in error. In actuality, the error is the incorrect functioning of the program over some subset of its input space. Two different programs in different programming languages can in this sense contain the same error -- they produce the same incorrect output without regard to the syntactic constructs that encode the error. Thus, a functional concept of error allows error analysis across programming languages, something difficult to achieve within a structural concept. Also, a functional view allows the correspondence of errors and program correctness to be clearly stated. To describe a program error in the functional sense means to describe a set of inputs that produce wrong results. With such a description it is possible to locate the structural construct that encodes the error with a good possibility of seeing how to correct it. The reverse is not true, however, since being told that given set of statements is wrong requires, in essence, the reconstruction of the functional error category from the specification to enable the error to be corrected.

To gain a better understanding of the various testing methodologies, it is useful to see what kinds of functional and structural errors each reveals. We have seen already that a useful structural categorization of errors is that of computational, domain, and case errors. Functional error categories have not been so clearly delineated, but may be inferred from the types of tests done in functional testing. If the competent programmer hypothesis applies to the function implemented as well as the code produced, we may conclude that functional errors occur as slight perturbations of the specification.

Errors in a function may be categorized as follows:

(1) Boundary conditions - The function may be incorrect on boundary points of the specified ranges of the input variables.

(2) Improper subfunction selection - The function may involve the computation of several subfunctions, some of which may be invoked at an improper time.

(3) Improper abstract relationships- The specification treats certain input variables as an abstraction. The function may group the wrong variables in attempting to implement the abstraction.

(4) Special values - Values like 0, 1, NULL often carry multiple meanings across data types. The function may be incorrect at these values.

It is important to note that these are errors in the sense of the implemented function (the input-output pairs) and not in the sense of the location in the program.

Clearly, no method is ideal for discovering all errors. Furthermore, every method specializes in finding particular errors. Thus, it is frequently suggested that a viable testing strategy is to combine several of the above structural and functional methods to achieve greater coverage of error categories. With more categories covered it is reasonable to assert that more errors will be discovered, thus increasing confidence in the correctness of the program.

5. Error Propagation and Elimination

To maximize error coverage in testing, much current research has focussed upon how to combine validation techniques that cover different error categories. One such combination is proposed here with examples. It involves a hybrid of verification and testing in which testing is used to establish the preconditions for a proof which essentially states that given an error, it will propagate to the output of the program.

Proposals for combining testing and formal verification have appeared several times [Goo75], [Ger76], and [Gel78]. Primarily the focus has been on how to simplify formal verification. The finite nature of testing suggests that testing could be used to prove the basis step for some of the inductive proofs necessary in formal verification [Goo75]. The difficult nature of theorem proving, suggests that only tested programs should be proved; the error prone nature of theorem proving suggests that all proved programs should be tested [Ger76]. Geller [Gel78] has attempted to combine testing and formal verification by using testing to simplify proofs. In this case, testing is used to verify cumbersome predicates, e.g. those involved in describing array initialization. The emphasis in all these approaches is that formal verification demonstrates the correctness of the program and testing supports this process.

The proposal of this paper is that testing and verification may be combined in quite another way, resulting in conclusions about the absence of certain errors in the program rather than the total correctness of the program. This is best explained by the following testing strategy:

(1) Identify the error categories of interest.

(2) Identify locations within the program where the errors could occur.

(3) For each potential error location:

    a. Derive a condition under which an error will be created at the given location (the <u>creation condition</u>).

b. Derive a condition under which an error will propagate to the end of the program (the <u>propagation condition</u>).

c. Produce the conditions of (a) and (b) with appropriate test data points, then inspect the output. If the output is correct, the potential error does not exist at the given location.

To clarify the above strategy, a precise description must be given for "error propagation."

5.1. Error Propagation

A comprehensive treatment of error propagation requires viewing the semantics of a program from a functional and structural perspective. In the functional approach [Lin79], a program is treated as a mathematical function, i.e. a set of input-output pairs. In the structural approach, a program is treated as a means of describing a set of computations. Informally, a computation is a trace of a program's execution. The set of computations of a program uniquely determines the program function, but not vice versa. The ordered input-output pairs of the program function bear no necessary correlation to the program variables. To provide this relationship, the concept of a "program state" is introduced.

<u>Definition</u> A <u>state</u> of a program P is a mapping

s: var --> value

which associates a unique value with every variable of P.
An <u>initial state</u> of a program is a state which exists before any statements of P have been executed. A <u>final state</u> is a state which exists after the program has halted.

Some variables $\{x_i\}$ of an initial state may be designated as program <u>input variables</u>. Their corresponding values $\{u_i\}$ are called the program <u>input</u>. Some variables $\{y_i\}$ of an initial state may be designated as program <u>output variables</u>. Their corresponding values $\{v_i\}$ upon program termination are called the program <u>output</u>. If no variables are explicitly designated as input or output, then all variables are considered both input and output

The <u>program function</u> that a program P computes, denoted by [P], is therefore,

$$[P] = \{(u,v) \mid v \text{ is the output of } P \text{ on input } u, \text{ where } u \text{ and } v \text{ are ordered sets of values}\}$$

Any arbitrary program segment P implicitly defines a program function with all variables designated input and output. For the purposes of this paper, "program" and "program segment" are used interchangeably unless otherwise stated.

Each approach to program semantics has certain advantages and disadvantages for error analysis (see Section 4.3). For error propagation, a functional semantics enables a clear definition of the conditions under which an error in an initial state will propagate to a final state.

<u>Definition</u> Let y be in the range of the function f. A <u>level set</u> of f is

$$D_y = \{x \mid f(x) = y\},$$

for some element y in the range of f.

<u>Definition</u> A <u>propagation condition</u> B of a function f, is a predicate defined on the domain of f satisfying the following:

For All $x <> y$ in dom(f),
$B(x)$ and $B(y)$ implies $f(x) <> f(y)$.

All domain elements of f which satisfy a propagation condition B produce different members of the range of f, i.e., they fall into different level sets of f. This is not to say that each pair of domain elements from different level sets necessarily satisfy B. It should also be noted that there may be more than one propagation condition for a function.

The concept of a propagation condition explains step (3) of the above strategy. Suppose a program P is divided into two parts, R and Q with [P] = [Q]o[R]. Suppose further that R and Q are correct and that R′ is an erroneous mutant of R. To ascertain that R is indeed the correct version and not R′, execute P on an arbitrary input s. If [R](s) can be shown to be different from [R′](s) and both [R](s) and [R′](s) satisfy a propagation condition for [Q], then the functional error from R′ will propagate through Q. If the *propagation condition* is satisfied, inspecting the output allows two conclusions. Not only is it known that P is correct for s (any testing strategy would have demonstrated this!), but it is also known that P′ (P with R′ replacing R) is not correct. If P′ were the correct version, P would have been incorrect on input s. Therefore, a potential error (that of substituting P for P′) has not occurred. If all potential errors were eliminated in this manner, the program would be proven correct for all input data.

A computational semantics is appropriate for analyzing the state transformations that occur as a program executes.

Definition A computation point for a program P is an ordered triplet, c = (n, i, s) where

n is a statement number of a statement in P,

i is the iteration count, the number of times that statement n in P has been executed.

s is a state of the program P.

This definition includes the iteration count to allow an isolated computation point to be identified with a particular statement as well as with a particular execution of that statement.

Definition A computation for a program P is a sequence of computation points representing the execution of P along any feasible path of P. A subcomputation for a program P is a subsequence of a computation of P.

A few comments are in order concerning the preceding definitions. First, the level of detail for computations could be increased. Computation points could contain the entire history of the execution of the program, including all the register loads, comparisons, etc. Second, a computation of a segment P′ of a program P is not necessarily a subcomputation of P. [P′] may be defined for inputs that may never occur as the result of earlier execution in P; therefore, P′ may have computations that do not occur as subcomputations of P. Third, computations are defined only for feasible paths. Non-feasible paths could never be executed and therefore have no corresponding computations.

Computations and computation points facilitate the discussion of error creation. Incorrect code must be executed for a functional error to occur. Thus, the computation for a functional error contains the information necessary to locate the error. An error may be "created" in one of two ways. First, an incorrect statement may produce an incorrect intermediate state. This state is incorrect in that the correct statement would have produced a different state. Second, the execution of an incorrect statement may lead to an incorrect successor statement being executed.

Any distinguishing characteristic of a correct computation may be used to decide if an arbitrary computation is incorrect. For instance, suppose that a final state is only obtainable by a computation of length greater than n. Any computation of length less than n may then be rejected as incorrect. The process of rejecting a computation may be viewed as applying a "characteristic function" to the computation. This function selects a subset of the computation points from the computation and then evaluates an expression on that subset. The following defines the format for two classes of characteristic functions.

<u>Definition</u> For any computation $C = (c_1, \ldots, c_j)$ for a program

(a) <u>exp@n</u> <u>on</u> <u>C</u> denotes the value [exp](s)

where s is the state of the last computation point for statement number n in C.

If a computation point for statement number n does not exist in C, then exp@n is undefined.

(b) <u>exp@hist(n)</u> <u>on</u> <u>C</u> denotes the k-tuple,

$([exp](s_1), [exp](s_2), \ldots, [exp](s_k))$

where

k is the number of occurrences in C of statement number n and $s_1, \ldots, s_k$ are obtained from the k computation points in C for statement number n.

If k = 0 then exp@hist(n) is undefined.

If k = 1 then exp@hist(n) = exp@n.

In the expressions exp@n and exp@hist(n), n and hist(n) are called the computation point specifiers, or more simply, the specifiers for the expression exp.

The specifiers are restricted to selecting either the last computation point or all computation points for a particular statement. This is because the primary concern is the effect a given statement has on a computation.

An error creation condition applied to a state of a computation point tells whether the succeeding computation point is in error. A creation condition is defined for a class of mutants of the correct construct. All states s which satisfy the creation condition are transcendental [Row81] for the class of mutants, i.e. when presented with a transcendental state s, no two mutants from the class transform s into the same state. For example, for the class of polynomials $P^+(M)$ with positive integral coefficients bounded above by M, any input value greater than M + 1 is transcendental [Row81]. If, therefore, an expected program error is the substitution of one member of this class for another, an error will be created whenever the polynomial is evaluated on a number exceeding M + 1. This error will be reflected in the next computation point if the value is assigned to a variable. If more computing is done first, as in the case of a comparison, this error may be canceled and have no effect on the next computation point. It is here that the detail of the computation impacts the detection of structural errors. A more detailed computation better distinguishes the instances of error creation and error cancellation.

Once an error has been introduced into a computation, it is then necessary to describe how the error will propagate to another part of the computation. To do this, it is sometimes desirable to relate functionally the values of expressions at two different points in the computation. If the second expression evaluates to an incorrect value whenever the first evaluates to an incorrect value, then any error reflected by the first expression will propagate to the second expression.

<u>Definition</u> Given a program P and class of computations S, for specifiers x and y,

exp1@x <u>influences</u> exp2@y <u>on</u> S

is used to mean the following:

```
For all C and D in S for which exp1@x and exp2@y
are both defined,
  if exp1@x on C <> exp1@x on D then
     exp2@y on C <> exp2@y on D.
```

If S is omitted, it is assumed to be the set of all computations of P.

A simple example will illustrate the idea of influence. Consider the code:

```
1  read (z);
2  read (y);
3  while y < 10 do
     begin
4      z := z + y;
5      y := y + 1
     end;
6  write (y, z)
```

For the class of computations S in which y@2 = 0 for the above code:

(1) z@1 influences z@4 implies that the output of the loop will be different for every input value of z.

(2) y@2 influences z@4 implies that the output of the loop will be different for every input value of y. This is trivially true for the class S, because the input y-value is constant for S.

(3) z@hist(3) influences z@4 implies that the output of the loop will be different for every sequence of z values the loop can compute.

(4) z@4 influences z@6 implies that an error in z upon loop exit will propagate to the output statement.

The influence dependences for a program cannot be determined algorithmically because this could require deciding if arbitrary loops halt, which is impossible (see Section 3). When such dependences can be shown, however, propagation conditions may be easier to prove. For example, suppose x@n1 influencas y@n2 and y@n2 influences z@n3 for a computation C. If it is known that x contains an incorrect value at line n1, then z contains an incorrect value at line n3, and the error has propagated.

In Section 4.1 a computation error was defined to be an incorrect computation along a particular path. To understand the propagation of a computation error, the following definitions are given.

Definition Let P be a program with computation C in which computation point c1 = (n1, i1, s1) precedes c2 = (n2, i2, s2). The intermediate code determined by c1 and c2 is the set of statements of P executed between n1 and n2 in the computation C. The intermediate function determined by c1 and c2 is the program function of the intermediate code.

Definition Let C = (c0, ... , cm, ... , cn) be a computation for a program P.

Let R be the intermediate code determined by c0 and cm.
Let Q be the intermediate code determined by cm and cn.
Suppose R is incorrect and that R* is a correct mutant of R.
Let s0 be a valid input for P for which [R] is defined.
Let

```
    [R](s0) = s1          [R*](s0) = s1*
    [Q](s1) = s2          [Q](s1*) = s2*
```

Let exp be any expression over the program variables.

(a) If s1 <> s1* then R has created a state error s1 for s1* on input s0.

(b) If R has created a state error and s2 <> s2*, the state error s1 for s1* propagates through Q.

(c) If [exp] (s1) <> [exp] (s1*) then R has created an expression error for exp.

The following theorems are trivially true by substitution of the definitions given earlier in this section. They are given here to illustrate how the concepts are related.

Theorem A state error s1 for s2 propagates through the intermediate code, P, iff s1 and s2 are in different level sets for [P].

Proof Consider state error s1 for s2. If s1 and s2 are in different level sets of [P], then [P] (s1) <> [P] (s2), and the state error propagates through P. If the state error propagates through P, then [P] (s1) <> [P] (s2). Thus, s1 and s2 are in different level sets of P.

Theorem Let P be the *intermediate code* delimited by statements n1 and n2. Let C and D be two computations of P with different initial states s_C and s_D, respectively.

(1) exp1@n1 influences exp2@n2 for all computations executing P,

iff

(2) If s_C, s_D are in different level sets of [exp1]

then [exp2][P] (s_J) <> [exp2][P] (s_k)

Proof Assume (1). If s_C and s_D are in different level sets of [exp1], then [exp1] (s_C) <> [exp1] (s_D). By definition of influence, [exp2] [P] (s_C) <> [exp2] [P] (s_D) and (2) follows immediately.
Assume (2). If [exp1] (s_C) <> [exp1] (s_D) then s_C and s_D are in different level sets of [exp1]. Combining this with (2) yields [exp2] [P] (s_C) <> [exp2] [P] (s_D), and (1) follows immediately by the definition of influence.

It has been noted already that there can be more than one propagation condition for a function f. This is clear because every propagation condition is implicitly defined by its propagation set, the set of all values that satisfy the propagation condition. Every subset of a propagation set is a propagation set so there are many propagation conditions.

Theorem Let B be a propagation set for a function f. No two members of B are in the same level set of f.

Proof Let x and y be different elements of B. By definition of a propagation set, f(x) <> f(y). Thus, x and y can not be in the same level set of f.

What characteristics, if any, qualify one propagation condition to be declared "better" than another? One characteristic is the "generality" of the propagation condition or set. Recall that a potential error can be eliminated only if a state error can be detected when it occurs. A state error s1 for s2 can be detected only if both s1 and s2 satisfy the propagation condition, i.e. both s1 and s2 are in the propagation set. Therefore, increasing the size of the propagation set increases the number of potential errors that can be eliminated. If B is any propagation set for the function f, it satisfies the following set equation:

$$B = \{x \mid \text{For All } y <> x \text{ in } B,\ f(x) <> f(y)\}.$$

Clearly, the smallest propagation set is the null set. Also by the last theorem above, all members of B are in different level sets of f. Since two propagation sets may both be infinite, describing one as larger than an other is inappropriate; the term "most general" may be used instead. Thus, a <u>most general</u> propagation set of a function f is a propagation set that contains one member from each level set of f. Associated with this set is a <u>most general</u> propagation condition.

A second characteristic of a propagation condition is that of applicability. Until now it has been implicitly assumed that the function will be applied to all domain elements. Since some values may not be feasible as input to an intermediate program function, the propagation set should contain as many feasible values as possible. Propagation conditions may therefore be compared on the basis of how many feasible values they contain.

A final characteristic of a propagation condition is that of efficiency. If two propagation conditions are equivalent in terms of applicability and generality, they may be differentiated on the basis of their ease of evaluation. This efficiency characteristic can not increase the number of errors that can be theoretically eliminated, but it may increase the number of errors that can be practically eliminated in an implementation.

5.2. <u>Testing Accumulation Programs</u>

Accumulation programs [Bas80] are the class of programs satisfying the following schema and restrictions:

```
1  Q:   z := z0;
2       while Not (y In Null(Y) ) do
          begin
3           z := acc (z, k(y));
4           y := h (y)
          end
```

<u>Restrictions</u>

(1) y denotes all program variables requiring definition for the loop body to be defined on any iteration. z does not enter into the computation for h(y) in line 4. Thus, z does not influence the flow of control of the loop.

(2) The functionalities of h, k, acc and Q are:

[h] : Y --> Y

[k] : Y --> Dbase

[acc] : Z x Dbase --> Z

[Q] : Z x Y --> Z x Y

where Y denotes the values the variable y may assume, Z denotes the values the variable z may assume, and Dbase denotes the range of [k]. Null(Y) denotes all values of y which terminate the loop.

The following is an example of an accumulation loop:

```
z := 0; y := 0;
while y < 10 do
  begin
    z := z + y;
    y := y + 1
  end
```

The variable z accumulates information as the loop iterates through different values of y. In this accumulation program we have,

Y, Z, Dbase = Natural Numbers

Null(Y) = {y | y <= 10}

[h] = successor function

[k] = identity function

[acc] = addition function

Clearly, the restrictions are satisfied.

In an accumulation loop, the restriction on z allows the computation for y to be separated from the computation for z. The above accumulation loop schema Q is therefore functionally equivalent to the following transformed schema QQ:

```
QQ: n := 0; z := z0;
    while Not (y In Null(Y) ) do
      begin
        Rh[n] := y;
        n := n + 1;
        y := h(y)
      end;
    j := 0;
    while j < n do
      begin
        z := acc (z, k (Rh[j]) )
        j := j + 1
      end
```

The first loop in QQ computes the same intermediate values of y that Q computes, but stores them in an array Rh (results of h). The second loop in QQ processes the array element-by-element, extracting the desired information (via k) and accumulating it into z (via acc).

Lemma The accumulation loop Q computes the function:

$[Q](z0, y0) = (z, y)$ such that

$z := ([acc]([acc](\ldots([acc](z0, [k](y0)), [k]([h](y0)), \ldots, [k]([h]^{n-1}(y0))$

$y := [h]^{n}(y0)$

where n is the smallest value such that $[h]^{n}(y0)$ is in Null(Y).

To be able to test accumulation loops for possible errors, it is necessary to understand how errors propagate through an accumulation loop. Suppose H is a class of mutants of h; we say that H is an error category for h. Clearly, a substitution of h' in H for h may affect the computation of the loop. If it can be shown that the substitution results in a "positive error" ($[h]'(y) > [h](y)$) during every iteration of the loop, and that this positive error accumulates into the variable z, then z will necessarily be incorrect on loop termination. The following theorem states sufficient conditions to guarantee that such accumulation does occur

Definition For two k-tuples,

$A = \{a_1, \ldots, a_k\}$ and $B = \{b_1, \ldots, b_k\}$,

$A <= B$ iff For all i, $a_i <= b_i$.

This definition is applied recursively if a_i and b_i are sets of the same cardinality. If $A <= B$ then B maximizes A

__Theorem__ For the accumulation program schema Q above, let

H denote the error category of h

Y denote the set of y-values for all possible iterations of the loop.

Z denote the set of z-values for all possible iterations of the loop.

D_k denote a subset of the domain of [k].

Assume Y, Z, and D_k each have a partial ordering operator, <=.

Let Q' result from substituting h' in H for h in Q. Let C be the computation of Q on input y0 and C' be the computation of Q' on input y0. The following conditions are sufficient to guarantee that an expression error for z occurs after executing Q and Q' on input y0.

(1) Q and Q' iterate the same number of times (> 1).

(2) For each iteration, [h](y) <= [h'](y),

or

For each iteration, [h](y) >= [h'](y).

(3) For at least one iteration, [h](y) <> [h'](y).

(4) [k] is strictly monotonic on D_k

(5) For each iteration both [h](y) and [h'](y) are members of D_k.

(6) [acc] is strictly monotonic in both variables.

__Proof__

Let Rh denote the tuple containing the initial and intermediate y-values computed by Q.

Let Rh' denote the tuple containing the initial and intermediate y-values computed by Q'.

By conditions (1) and (2) of the theorem, assume without loss of generality that Rh' maximizes Rh.

By conditions (3), (4) and (5), [k](Rh') maximizes [k](Rh).

Let

$[k](Rh) = (z_1, \ldots, z_n)$ and $[k](Rh') = (z_1', \ldots, z_n')$.

For schema Q (by the preceding lemma),

$$z = [acc]([acc](\ldots ([acc](z_0, z_1), z_2), \ldots), z_n)$$

and for schema Q',

$$z = [acc]([acc](\ldots ([acc](z_0', z_1'), z_2'), \ldots), z_n')$$

By repeated application of condition (6) of the theorem, we may conclude that an expression error occurs for z.

In the following example, the above theorem will be used to aid in testing a program containing an accumulation loop.

5.3. Example

The strategy and the theory developed are now applied to a program which computes the area under a curve by rectangular approximation. Test data is not included due to the generality of the polynomials in the program.

```
program calcarea (input, output);
  var a, b, incr, area, value : real;
  begin
  1      read (a,b,incr);   {incr > 0}
  2      value := p1(a) ;
  3      area  := 0;
  4      while a + incr <= b do
            begin
  5              area := area + value * incr;
  6              a := a + incr;
  7              value := p2(a)
            end;
  8      incr := b - a;
  9      if incr >= 0 then begin
 10         area := area + value * incr;
 11         writeln ('area by rectangular method:', area)
         end else
 12         writeln ('illegal values for a=', a, ' and b=', b)
  end.
```

Using the strategy given above, we have the following.

<u>Step 1</u> <u>Identify the error categories</u>.

(1) Incorrect polynomials p1 and p2, both of which are members of $P^{+}(M)$, the set of polynomials of the form

$$a_0 + a_1x^1 + \dots + a_nx^n$$

where

$0 <= a_i <= M$ and a_i is an integer.

$P^{+}(M)$ is an important category of polynomials for which transcendental testing is appropriate [Row81]. (See Section 5.1. of this paper.)

(2) Incorrect comparisons. A wrong comparison operator is used.

(3) Incorrect accumulation. Wrong variables or wrong operators are used.

(4) Incorrect initialization.

(5) Incorrect output. An incorrect variable is used in an output statement.

<u>Step 2</u> <u>Identify the locations where errors could occur</u>.

<u>Error</u>	<u>Lines</u>
1. Incorrect Polynomials (IP)	2, 7
2. Incorrect Comparisons (IC)	4, 9
3. Incorrect Accumulation (IA)	5, 6, 10
4. Incorrect Initialization (II)	2, 3, 8
5. Incorrect Output (IO)	11, 12

<u>Step 3</u> <u>Derive Creation and Propagation Conditions</u>

A creation condition and propagation condition are now provided for each of the error locations given in step 2. A creation condition guarantees that the error, if present, produces a state error. It must be satisfied just before the potentially erroneous statement is executed. A propagation condition guarantees that this state error propagates to an output statement. It is evaluated immediately after the potentially erroneous statement is executed.

Each error is designated by an error category abbreviation, followed by a line number, e.g., IP 7 designates "Incorrect Polynomial at line 7."

<u>Error IO 11, 12</u>

<u>Creation Condition</u> -- all variables have different values.

<u>Propagation Condition</u> -- true

<u>Error IA 10</u>

Two cases are considered.

(1) An incorrect accumulation operator has been used; perhaps + should have been -, *, or /.

Error	Creation Condition
-	value * incr <> 0
*	area > 2 and value * incr > 2
/	value * incr > 1 and area > 1

Propagation Condition -- true.

(2) An incorrect accumulation base element (value * incr) has been used.

Error	Creation Condition
Off by a constant	true
Off by a factor	value * incr <> 0

Propagation Condition -- true.

Error IC 9

Perhaps the >= should have been another comparison operator.

Error	Creation Condition
>	incr = 0
<	true
<=	incr <> 0
=	incr > 0
<>	incr <= 0

Propagation Condition -- true.

Error II 8

Three cases are considered.

(1) The wrong variable may occur on the left hand side of the assignment. This error is particularly nasty because any resulting functional errors depend upon whether the incorrect variable is used ("live") or not used ("dead") in the remaining computation. If the incorrect variable is dead, then a functional error can occur only when the correct variable influences the output of the program. Data flow analysis will detect this type of error. Additionally, if the incorrect variable is live, then a functional error will also occur whenever the incorrect variable influences the output of the program. Since data flow analysis can isolate the first error, the second error is considered here.

Error	Creation Condition
Substitution of a live variable for incr on the left side of the assignment statement	[incr <> b-a]@4

Propagation Condition

Two propagation conditions are given, the first representing a domain error and the second representing a computation error.

[incr >= 0 and (b-a) < 0]@4

[incr >= 0 and value <> 0]@4

Recall that the specifier @4 implies evaluation of these expressions at loop termination. These two conditions may be combined yielding:

[incr >= 0 and ((b < a) or value <> 0)]@4

(2) A wrong variable may have been substituted on the right hand side of the assignment.

Error	Creation Condition
variable 'a' is an incorrect variable	All variables have different values from 'a'.
variable 'b' is an incorrect variable	All variables have different values from 'b'.

Propagation Condition -- Same as in case (1).

(3) An incorrect constant expression may have been used on the right hand side of the assignment.

Error	Creation Condition
Off by a constant	true
Off by a factor	[b-a <>0]@4

Propagation Condition -- Same as in case (1).

Error IP 7

Creation Condition

a@6 > M + 1 for each iteration of the loop and the loop executes more than once.

Recall that all values greater than M + 1 are transcendental for all polynomials in $P^{+}(M)$. See Section 5.1 of this paper.

Propagation Condition -- true.

The simplicity of this propagation condition is guaranteed by the accumulation loop theorem from the previous section. In order to show this, it must be shown that the loop is an accumulation loop and that it satisfies the conditions of the theorem. If this is the case, then the theorem states that there will be an expression error for area on loop exit. Furthermore, on the last iteration of the loop, a + incr <= b on loop entry, so b - a >= 0 on loop exit. But, b - a >= 0 is the propagation condition which ensures that area@8 influences area@10; any error in value@10 merely increases the magnitude of the error in area@10. Thus, if the loop satisfies the conditions of the theorem, the given error will propagate to the output statement in line 11.

To show that the loop is an accumulation loop, we note the following correspondences to the schema G:

y corresponds to (a, b, incr, value)
Null(Y) = {(a, b, incr, value) ! a + incr > 0}
z corresponds to area
[h](a, b, incr, value) = (a+incr, b, incr, p2(a+incr))
[k](a, b, incr, value) = value * incr
[acc](area, x) = area + x

Clearly, area does not enter into the computation of h. Also,

Y = Real x Real x Real x Real
Z = Real
Dbase = Real
D_k = Real x Real x {incr} x Real, i.e. D_k has a fixed value of incr.

To show that the loop satisfies the conditions of the theorem, we first note that the error category is

H = {h' ! [h]'(a, b, incr, value) = (a+incr, b, incr, [P'](a+incr))},
where P' is a member of $P^{+}(M)$.

The theorem conditions are therefore satisfied.

(1) Substitution of h' (from error category H) for h does not change the number of times the loop executes, since the computation for a, incr, and b remain unaffected.

(2) Provided the creation condition is satisfied for all iterations of the loop,

[h'](y) >= [h](y) for each iteration
or
[h](y) >= [h'](y) for each iteration.

If this were not the case, then for some t1 and t2, transcendentals for h and h',

[h](t1) > [h'](t1) and [h'](t2) > [h](t2).

Since the functions [h] and [h'] are continuous, there must be a point t between t1 and t2 such that [h](t) = [h'](t). Thus, t is not a transcendental. But this is a contradiction since all points greater than M + 1 are transcendental and t > t1 > M + 1. Thus the functions [h] and [h'] do not cross on any point beyond M + 1 and one always maximizes the other on this interval.

(3) If the creation condition is true, [h'](y) <> [h](y) for all iterations of the loop.

(4) [k] is strictly monotonic on D_k since incr is a nonzero constant for the loop.

(5) For all iterations [h] and [h'] produce members of D . This is clear because all members of H vary only in their computation for value, leaving the computation a, b, and incr unaffected.

(6) [acc] is strictly monotonic in area and value*incr.

Error IA 6

A wrong h function may be implemented. Two instances are considered.

Error	Creation Condition
Off by a constant > 0	true
Off by a factor > 1	incr > 0

Propagation Condition -- true

For either error, a > b upon loop exit, as can be seen from the last iteration of the loop. Thus, [incr < 0]@8 causes a domain error with statement 12 executed in place of statement 11.

Error IA 5

A wrong k function may be implemented.

Error	Creation Condition
Off by a constant	true
Off by a factor	value * incr <> 0

Propagation Condition -- true.

Since neither of these two errors affect the monotonicity of k, the argument used for IP 7 holds.

Error IC 4

In place of the <=, another comparison may have been substituted.

Error	Creation Condition
<	a + incr = b
>	true
>=	a + incr <> b
=	a + incr < b
<>	a + incr >= b

Propagation Condition

The substitution of < for <= is an excellent example of a mutant being unobviously equivalent to the given program. This is discovered while attempting to find the propagation condition for this "error." The substitution may cause the loop to halt one iteration too soon, with termination guaranteeing that incr is unchanged by line 8. Thus, the execution of line 10 computes the same value for area as an additional execution of line 5. An additional execution of the loop would result in [incr = 0]@8, so [value * incr= 0]@9. Hence, statement 10 would not change the value of area computed by the additional execution of the loop. Thus, the substitution of < for <= is an equivalent mutant and the propagation condition is false.

For the other four substitutions to influence an output, a created error must propagate to statement 11 or 12. A propagation condition of a + 2*incr < b is sufficient, since this guarantees the loop will execute at least twice, causing the loop computed value for area to be strictly greater than the tail approximation in lines 8-10.

Error II 3

Suppose area should have been initialized to another constant.

Creation Condition -- true

Propagation Condition -- a <= b

With this condition satisfied, area@3 influences area@10. Therefore, if an error has occurred at line 3, an incorrect value for area will be printed by line 10.

Error IP 2, II 2

Creation Condition -- a > M + 1

Propagation Condition -- $a < b < a + incr$

This condition forces the program to follow the path,

$p = (1, 2, 3, 4f, 8, 9t, 10, 11)$,

skipping the loop body. [a < b]@1 ensures that [incr > 0]@9, so value@10 influences area@10. Since value@10 = value@2 for path p, value@2 influences area@10.

5.4. Applying the Strategy

It should be noted that errors have been eliminated in a "bottom-up" fashion. Recall that the justification for the strategy assumed that a program could be separated into two segments R and Q, with Q being correct. Certainly if R is the whole program, Q is trivially correct. As errors are eliminated from the end of R, then Q can expand to contain this "correct" code. It is possible, however, that R contains two structural errors that mask one another, with the first preventing discovery of the second on certain paths, and vice versa. For example,

```
1    x := 3 * y;
       .
       .
       .
2    z := x - 4;
3    write (z);
```

Suppose the error category of interest is "incorrect constants." Clearly, both the creation and propagation condition are true for this error in lines 1 and 2; any state will produce an incorrect state and the error is guaranteed to propagate to line 3. Yet, for y@1 = 4, the following program is equivalent and contains the incorrect constants:

```
1    x := 2 * y;
       .
       .
       .
2    z := x;
3    write (z);
```

Testing additional paths in which statements 1 and 2 are not coupled, and testing a coupled path with more inputs are two ways of reducing the impact of such errors. The situation is similar to that of linear domain testing in which assignment and equality blindness prevent certain predicate errors from being eliminated. Here, however, the blindness is due to a presumed error in the first part of the program, rather than in the correct operation of the first part of the program. Evidence exists that such coupling rarely occurs in practice [Bud80], but investigation of the phenomenon may yield greater insight into error propagation. Such investigation is currently under way.

6. Conclusions

An error-based testing strategy has been proposed for combining testing and verification in a new way. Some of the advantages are as follows:

(1) In structural and functional testing incorrect code may remain undetected even though executed. Consequently, the certainty of the results is difficult to determine. The strategy given here can guarantee that certain common errors are not present in the program. Structural and functional testing, on the other hand, only guarantee the elimination of very few error categories. Test data that guarantees the elimination of certain errors in addition to satisfying the usual functional and structural criteria is necessarily of better quality than test data that issues no guarantee. Thus, the proposed strategy provides a means of increasing test data quality.

(2) When quality is lacking, the tester can be directed to specific lines of code where potential errors have not yet been eliminated. This guidance is more specific than possible with structural testing.

(3) The proposed strategy is more efficient than mutation testing for killing particular mutants. First, the actual mutants do not have to be generated or executed. Second, one test point can eliminate all mutants along an execution path. Third, mutation testing provides little guidance when a mutant executes correctly. The proposed strategy guides the data selection process towards selecting data that creates the state in which an error could occur and in which it will propagate.

(4) The proposed strategy is based upon the function testing suggested by Foster and Howden, from which the concept of a creation condition may be inferred. The inclusion of a propagation condition in the strategy providas greater assurance that created errors will not be canceled by the remaining execution of the program.

(5) The proposed strategy extends linear domain testing by removing two restrictions. First, it need not be assumed that the

predicates to be tested are a linear combination of the input variables. Second, it need not be assumed that a domain error necessarily produces a functional error. Indeed, this must be proven in the proposed strategy.

(6) The proposed strategy combines testing and formal verification in a new way. The goal is to force the program to inform the tester of its own errors through testing. Formal verification is used to support this process. As a support tool, formal verification is used in a restricted capacity, lessening the difficulty normally encountered in formal proofs of correctness.

(7) Weyuker [Wey81] has argued that a testing strategy should use all the information that can be obtained from the program, the program's specification, and the errors commonly encountered. This strategy suggests that the computation of the program itself is another important source of information. The wealth of information that is contained in the computation has been virtually untapped by structural testing. The computation of a program on one input effectively eliminates a huge number of possible errors. Greater knowledge of these eliminated errors would increase our confidence in a program's correctness.

One weakness of the proposed strategy is the assumption shared with mutation testing that errors can be eliminated one at a time; i.e. two errors do not interact in such a way that each error prevents the other error from being eliminated by the strategy. There is evidence that this "coupling effect" rarely occurs in practice [Bud80], yet the strategy does not currently handle such situations. The concepts of creation conditions, propagation conditions, and influence do provide a framework in which such errors can be discussed and analyzed. Such work is currently in progress.

Point (7) suggests the direction of research needed.

(1) Program errors need to be categorized and creation conditions developed for each category.

(2) Loops other than accumulation loops need to be analyzed as to their error propagation characteristics.

(3) Automatic methods for developing propagation conditions need to be developed.

(4) Methods need to be developed for correlating the information obtained from different computations. A set of computations may collectively eliminate an error category for which no individual computation in the set can. For example, consider the potential error in which an incorrect variable occurs in an output statement. The creation condition of "all variables different from X" may not be satisfied on any one computation, but over a set of computations all variables may indeed be differentiated from X. This potential error can therefore be eliminated by the collective evidence from the set of computations.

REFERENCES

[Bas80] Basu, Sanat K., "A Note on The Synthesis of Inductive Assertions," IEEE TSE SE-6, 1, pp. 32-39 (Jan. 1980).

[Bud80] Budd, Timothy A., DeMillo, Richard A., Lipton, Richard J., and Sayward, Fredrick G., "Theoretical and Empirical Studies on Using Program Mutation to test the Function Correctness of Programs," POPL, pp. 220-233 (1980).

[Cla77] Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE TSE SE-2, pp. 215-222 (Sept. 1977).

[DeM78] DeMillo, R. A., Lipton, R. J., and Sawyer, F. G., "Hints on Test Data Selection: Help for the Practicing Programmer," Computer 11, pp. 34-41 (April 1978).

[Fos78] Foster, K., "Error Sensitive Test Analysis (ESTA)," Digest for the Workshop on Software Testing and Test Documentation, (Dec. 1978).

[Gel78] Geller, M., "Test Data aws an Aid in Proving Program Correctness," CACM 21, pp. 368-375 (May 1978).

[Ger76] Gerhart, Susan L. and Yelowitz, Lawrence, "Observations of Fallibility in Applications of Modern Programming Methodologies," IEEE TSE SE-2, 3, (Sept. 1976).

[Goo75] Goodenough, John B. and Gerhart, Susan L., "Toward a Theory of Test Data Selection," IEEE Trans. Soft. Eng. TSE-SE1, 2, pp. 156-173 (June, 1975).

[Ham74] Hamlet, Richard, Introduction to Computation Theory, Intext (1974).

[Ham77] Hamlet, Richard G., "Testing Programs with the aid of a Compiler," IEEE TSE SE-3, 4, (July, 1977).

[Han76] Hantler, Sidney L. and King, James C., "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys 8, 3, pp. 331-353 (Sept. 1976).

[Hen77] Hennie, F., Introduction to Computability, Addison-Wesley (1977).

[How76] Howden, William E., "Reliability of the Path Analysis Testing Strategy," IEEE TSE SE-2, 3, (Sept. 1976).

[How77] Howden, William. E., "Symbolic Testing and the DISSECT Symbolic Evaluation System ," IEEE TSE SE-3, pp. 266-278 (1977).

[How78] Howden, William. E., "A Survey of Dynamic Analysis Methods," pp. 184-206 in *Tutorial*: *Software Testing and Validation Methods*, ed. E. Miller, (1978).

[How78b] Howden, William E., "Algebraic Program Testing," *Acta Informatica 10*, (1978).

[How80b] Howden, William E., "Functional Program Testing," *IEEE TSE SE-6*, 2, pp. 162-169 (March 1980).

[How80] Howden, William E., "Completeness Criteria for Testing Elementary Program Functions," DM-212-IR, Dept. of Mathematics, University of Victoria, Victoria, British Columbia (1980).

[Lin79] L[illegible]er, R. C., Mills, H. D., and Witt, B. I., *Structured Programming Theory and Practice*, Addison-Wesley (1979).

[Mey67] Meyer, A. R. and Ritchie, D. M., "The Complexity of Loop Programs," *Proceedings of the ACM National Meeting*, pp. 465-469 (1967).

[Mil74] Miller, E., Paige, M., Benson, J., and Wisehart, W., "Structural Techniques of Program Validation," *Digest of Papers COMPCON 74*, pp. 161-164 (Spring 1974).

[Ost76] Osterweil, L. J. and Fosdick, L. D., "DAVE -- A validation Error Detection and Documentation System for Fortran Programs," *Software Practice and Experience 6*, pp. 473-486 (1976).

[Row81] Rowland, John H. and Davis, Philip J., "On the Use of Transcendentals for Program Testing," *JACM 28*, 1, pp. 181-190 (Jan. 1981).

[Tsi70] Tsichritzis, D., "The Equivalence Problem of Simple Programs," *JACM 17*, 4, pp. 729-738 (Oct. 1970).

[Wey80] Weyuker, Elaine J. and Ostrand, Thomas J., "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE TSE SE-6*, 3, pp. 236-246 (May 1980).

[Wey81] Weyuker, Elaine J., "An Error-Based Testing Strategy," New York University Report, Department of Computer Science, Courant Institute of Mathematical Science (Jan. 1981).

[Zei80] Zeil, Steven J. and White, Lee J., "Sufficient Test Sets for Path Analysis Testing Strategies," OSU-CISRC-TR-80-6, Computer and Inforamtion Science Research Center The Ohio State University (1980).